AIMLZG537- Information Retrieval

Dr. Maheswari Karthikeyan

Lecture5

BITS Pilani
Pilani Campus

Index construction
- T1, Chapter 4

Slides are adapted from Introduction to Information Retrieval;Manning,Raghavan

http://nlp.stanford.edu/IR-book

Chapter 4: Index construction

- Hardware Basics
- Blocked sort-based Indexing
- Single pass in-memory indexing
- Distributed and Dynamic indexing

Hardware Basics

- **When building an information retrieval (IR) system, many decisions are based on the characteristics of the computer hardware on which the system runs**.
  - Access to data in memory is ***much*** faster than access to data on disk.
    - Caching
  - Disk seeks: No data is transferred from disk while the disk head is being positioned.
  - Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
  - Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
  - Block sizes: 8KB to 256 KB.
  - Compressed data read
  - Servers used in IR systems typically have several gigabytes (GB) of main memory, sometimes tens of GB. Available disk space is several orders of magnitude larger

Blocked sort-based indexing

***Reuters-RCV1*:**

- Shakespeare's collected works are not large enough to demonstrte scalable index construction algorithms.
- Instead, we will use the Reuters RCV1 collection.
- English newswire articles published in a 12-month period (1995/6)

A Reuters RCV1 Document

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

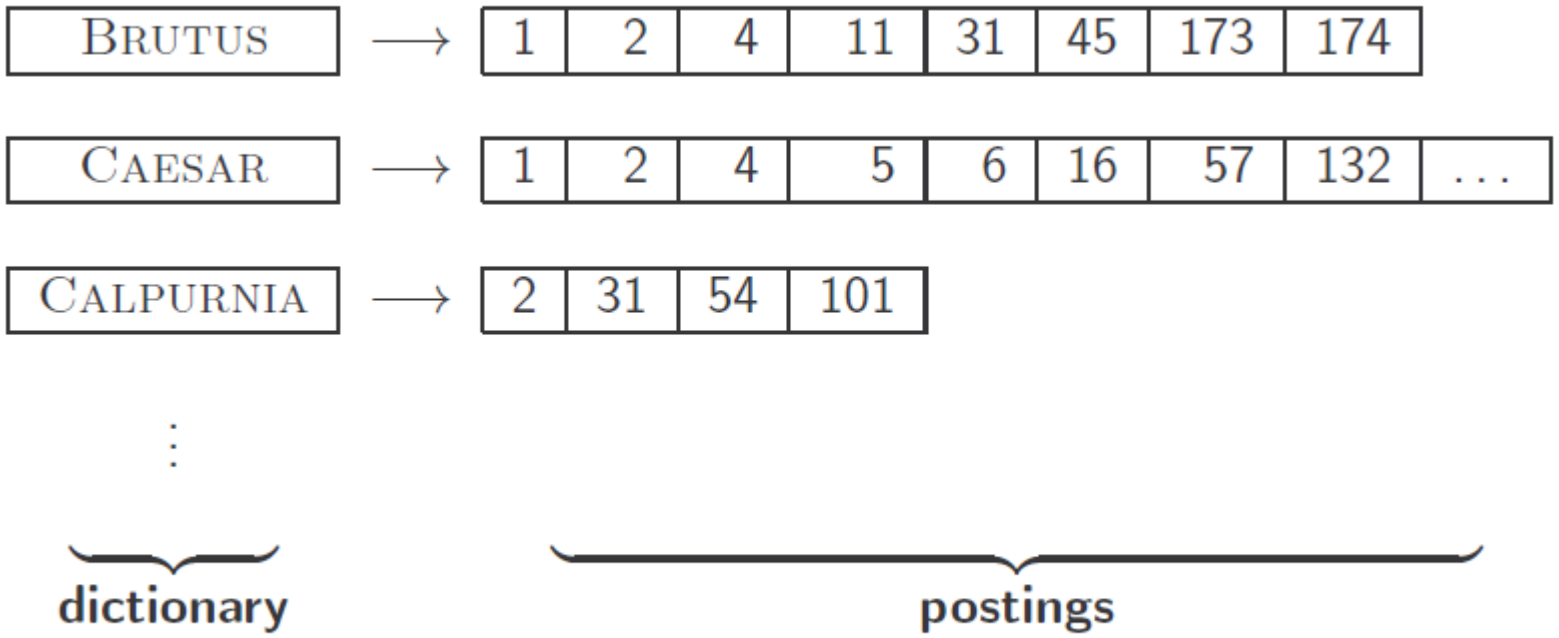
SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Reuters RCV1 Statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
	non-positional postings	100,000,000

Inverted Index construction



Inverted Index construction

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Sort-based index construction

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- But for large collections, we cannot keep all postings in memory and do the sort in-memory at the end
- We cannot sort very large sets of records on disk either (too many disk seeks)
- Need to store intermediate results on disk.
- Need an external sorting algorithm.

Sort-based index construction

- We must sort T = 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, term frequency).
- Define a block to consist of 10,000,000 such postings
  - Can easily fit that many postings into memory.
  - 10 such blocks for RCV1.
- Basic idea of algorithm:

For each block: (i) accumulate postings, (ii) sort in memory, (iii) write to disk

- Then merge the blocks into one long sorted order.

Blocked sort-based indexing (BSBI)

- Segments the collection into parts of equal size
- Sorts the termID – docID pair of each part in memory
- Stores intermediate sorted results on disk
- Merges all intermediate results into the final index.

BSBI Index Construction Algorithm

innovate achieve lead

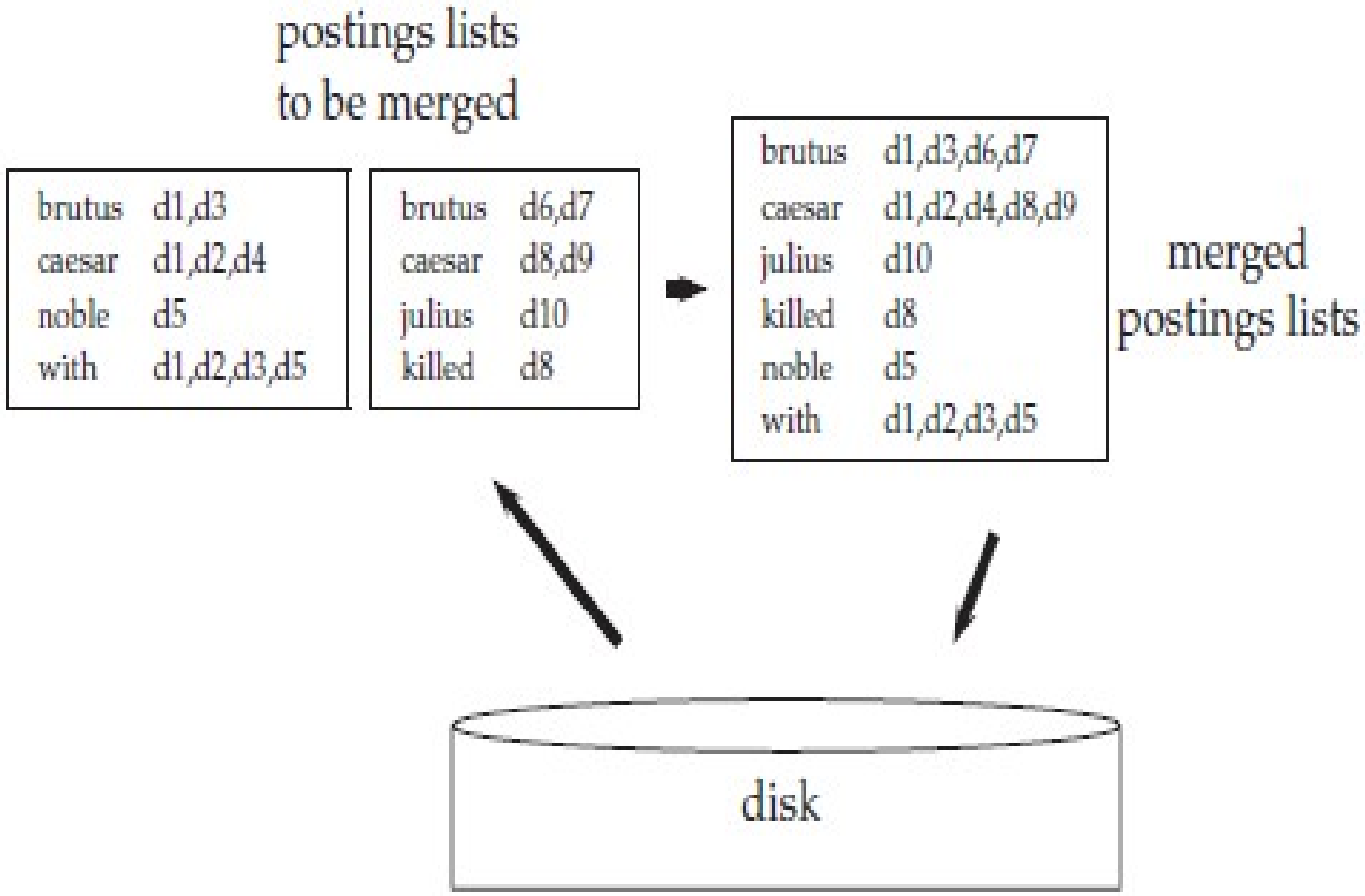
```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4      block ← ParseNextBlock()
5      BSBI-Invert(block)
6      WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

BSBI Index Construction Algorithm

innovate achieve lead

- **PARSENEXTBLOCK** --- The algorithm parses documents into termID–docID pairs and accumulates the pairs in memory until a block of a fixed size is full
- The block is then inverted and written to disk.
- Inversion involves two steps. First, sort the termID–docID pairs. Next, collect all termID–docID pairs with the same termID into a postings list, where a *posting* is simply a docID.

MERGEBLOCKS - Example



BSBI Algorithm

innovate achieve lead

- **Excellent scaling properties**
- We need a data structure in order to implement a term to termID mapping (which grows dynamically)
- For very large collections, this data structure does not fit into memory
- We would end up with a scalable, but very slow index construction method
- More Scalable alternative is

Single-pass in-memory indexing (SPIMI)

Single-Pass In-Memory Indexing (SPIMI)

- SPIMI uses terms instead of termIDs
- Writes each block's dictionary to disk
- Starts a new dictionary for the next block
- Can index collections of any size as long as there is enough disk space available.

Single-Pass In-Memory Indexing (SPIMI)

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur.

Single-Pass In-Memory Indexing (SPIMI)

innovate achieve lead

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Single-Pass In-Memory Indexing (SPIMI)

innovate achieve lead

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of posting

Distributed Indexing

- Data Collection is too big that index construction cannot be performed efficiently on a single machine
- Especially for web-scale indexing it must use a distributed computing cluster.
- Web search engines, therefore, use *distributed indexing* algorithms for index construction.
- The result of the construction process is a distributed index that is partitioned across several machines – either according to term or according to document.

Distributed Indexing

innovate achieve lead

- We need large computer *clusters* to construct any reasonably sized web index.
- Clustering is to solve large computing problems on cheap commodity machines or *nodes* that are built from standard parts (processor, memory, disk)
- Individual machines are fault-prone - Can unpredictably slow down or fail

Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world.

Google Data Centre Tour

M45: Open Academic Cluster

Yahoo M45 cluster:

- Datacenter in a Box (DiB)
- 1000 nodes, 4000 cores, 3TB RAM,
- 1.5PB disk
- High bandwidth connection to Internet
- Located on Yahoo campus
- World's top 50 supercomputer

Large scale computing

Challenges:

- How do you distribute computation?
- Distributed/parallel programming is hard
- Machines fail

Map-reduce addresses all the above

- Google's computational/data manipulation model
- Elegant way to work with big data

Distributed Indexing

- The distributed index construction method described is an application of ***MapReduce***, a general architecture for distributed computing
- MapReduce is designed for large computer clusters.

Distributed indexing

- Maintain a *master* machine directing the indexing job .
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns/reassigns each task to an idle machine from a pool.

MapReduce

innovate achieve lead

Two Phases:

- Map and Reduce
  - split up the computing job into chunks that standard machines can process in a short time.

Steps in MapReduce Process:

- The input data, is split into *n splits* where the size of the split is chosen to ensure that the work can be distributed evenly and efficiently
- The *map phase* of MapReduce consists of mapping splits of the input data to key-value pairs.
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

Map Phase-Parsers

- Master assigns a split to an idle parser machine
- Each parser writes its output to local intermediate files, the ***segment files***
- Parser writes pairs into *j* partitions
- Each partition is for a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***) – here $j = 3$.
  - Each term partition thus corresponds to *r* segments files, where *r* is the number of parsers

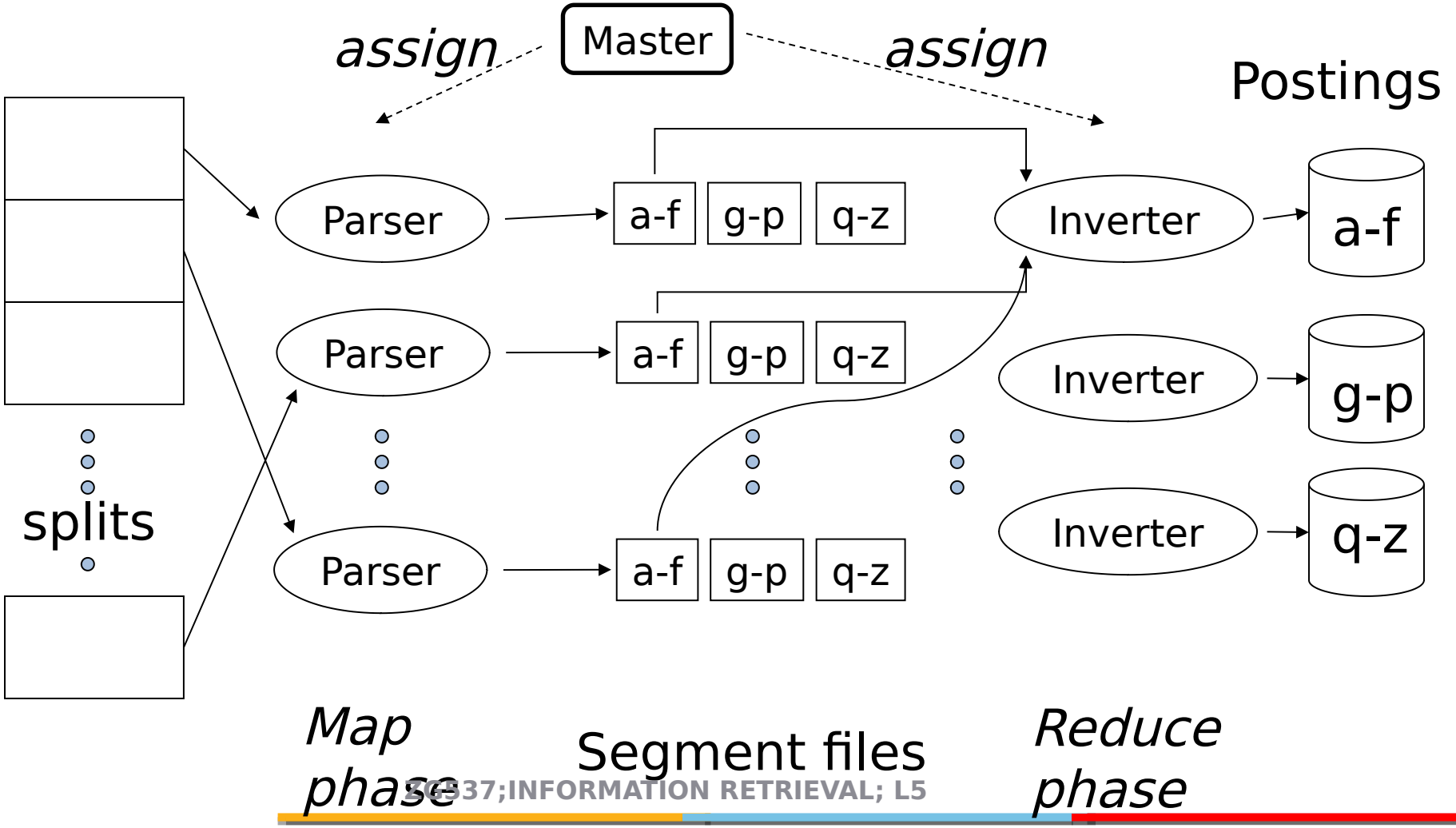
Reduce phase - Inverters

- An inverter collects all (term,doc) pairs

(= postings) for one term-partition

- Sorts and writes to postings lists.

Data flow


assign
Master
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Segment files
Reduce phase

MapReduce

- The index construction algorithm we just described is an instance of MapReduce
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing without having to write code for the distribution part
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

MapReduce

- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
  - ***Term-partitioned:*** one machine handles a subrange of terms
  - ***Document-partitioned:*** one machine handles a subrange of documents
- Most search engines use a document-partitioned index, better load balancing, etc.

Schema for index construction in MapReduce

Schema of map and reduce functions

map: input → list(k, v) reduce: (k,list(v)) → output

Instantiation of the schema for index construction

map: web collection → list(termID, docID)

reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …)
→ (postings list1, postings list2, …)

Example for index construction

map: d2 : C died. d1 : C came, C c'ed. **→ (<C, d2>, <died,d2>, <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>**

reduce: (<C,(d2,d1,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) → **(<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)**

Dynamic Indexing

- So far, we have assumed that the document collection is static, fine for collections that change infrequently or never (e.g., the Bible or Shakespeare).
- Most collections are modified frequently with documents being added, deleted, and updated.
    - new terms need to be added to the dictionary,
    - postings lists need to be updated for existing terms.

Simplest approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Issues with main and auxiliary indexes

- Problem of frequent merges
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for O/S.
- Assumption : The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

Logarithmic merge

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, …) on disk
- If Z_0 gets too big ($> n$), write to disk as I_0
- or merge with I_0 (if I_0 already exists) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
- Or merge with I_1 to form Z_2
- etc.

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 2  if |Z_0| = n
 3     then for i ← 0 to ∞
 4          do if I_i ∈ indexes
 5                then Z_{i+1} ← MERGE(I_i, Z_i)
 6                     (Z_{i+1} is a temporary index on disk.)
 7                     indexes ← indexes − {I_i}
 8                else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
 9                     indexes ← indexes ∪ {I_i}
10                     BREAK
11          Z_0 ← ∅
```

```
LOGARITHMICMERGE()
1  Z_0 ← ∅     (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

Further issues with multiple indexes

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
    - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
    - One possibility: ignore everything but the main index for such ordering

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted

Other sorts of indexes

- **Positional indexes**
  - Same sort of sorting problem … just larger
- **Building character n-gram indexes:**
  - As text is parsed, enumerate *n*-grams.
  - For each *n*-gram, need pointers to all dictionary terms containing it – the "postings".
  - Note that the same "postings entry" will arise repeatedly in parsing the docs – need efficient hashing to keep track of this.
    - E.g., that the trigram *uou* occurs in the term ***deciduous*** will be discovered on each text occurrence of ***deciduous***
    - Only need to process each term once

Exercise 0.46

For $n = 2$ and $1 \leq T \leq 30$, perform a step-by-step simulation of the algorithm in Figure 4.7. Create a table that shows, for each point in time at which $T = 2 * k$ tokens have been processed ($1 \leq k \leq 15$), which of the three indexes $I_0, \ldots, I_3$ are in use. The first three lines of the table are given below.

	I_3	I_2	I_1	I_0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0

SOLUTION.

	I_3	I_2	I_1	I_0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0
8	0	0	1	1
10	0	1	0	0
12	0	1	0	1
14	0	1	1	0
16	0	1	1	1
18	1	0	0	0
20	1	0	0	1
22	1	0	1	0
24	1	0	1	1
26	1	1	0	0

Google Data Center 360 degree Tour

https://www.sunbirddcim.com/video/google-data-center-360%C2%B0-tour

Resources

Chapter 4 of IIR

Original publication on MapReduce: Dean and Ghemawat (2004)

Original publication on SPIMI: Heinz and Zobel (2003)

41-30 mins https://www.youtube.com/watch?v=dH0pEySEoQo

42-13 mins WDM 42: Index Construction Using InMemory Sorting - YouTube

43- 43 mins

44-12 mins(repeated part of 43) no need to watch 45-10 mins(repeated part of 43) no need to watch

46-25 mins (skip 8 mins) https://www.youtube.com/watch?v=WHS9f_wbU-E
47-QA of 46 not required WDM 47: Q & A on Distributed Index – YouTube
48 – 24 mins WDM 48: Map Reduce – YouTube

49 – 11 mins WDM 49: Dynamic indexing using naive approach - YouTube 50-17 mins WDM 50: Dynamic indexing using logarithimic merge – YouTube
51- 3 minutes WDM 51: Issues With Multiple Indexes - YouTube

Index Compression (T1, Chapter 5)

Slides are adapted from Introduction to Information Retrieval;Manning,Raghavan

http://nlp.stanford.edu/IR-book

Need for Compression

- **Use less disk space**
  - Saves a little money
- **Keep more stuff in memory**
  - Increases speed
- **Increase speed of data transfer from disk to memory**
  - [read compressed data | decompress] is faster than [read uncompressed data]
  - Premise: Decompression algorithms are fast
    - True of the decompression algorithms we use

Why compression for inverted indexes?

Dictionary

- Make it small enough to keep in main memory
- Make it so small that you can keep some postings lists in main memory too

Postings file(s)

- Reduce disk space needed
- Decrease time needed to read postings lists from disk
- Large search engines keep a significant part of the postings in memory.
  - Compression lets you keep more in memory

- We will devise various IR-specific compression schemes

Reuters-RCV1:

- Shakespeare's collected works are not large enough to demonstrate scalable index construction algorithms.
- Instead, we will use the Reuters RCV1 collection.
- English newswire articles published in a 12-month period (1995/6)

A Reuters RCV1 Document

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Reuters RCV1 Statistics

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	~400,000
	avg. # bytes per token (incl. spaces/punct.)	6
	avg. # bytes per token (without spaces/punct.)	4.5
	avg. # bytes per term	7.5
	non-positional postings	100,000,000

Index parameters vs. what we index
(details *IIR* Table 5.1)

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	Δ %	cumul %	Size (K)	Δ %	cumul %	Size (K)	Δ %	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Lossless vs. lossy compression

innovate achieve lead

Lossless compression: All information is preserved
- What we mostly do in IR

Lossy compression: Discard some information

- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.

Vocabulary vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Can't : At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size.

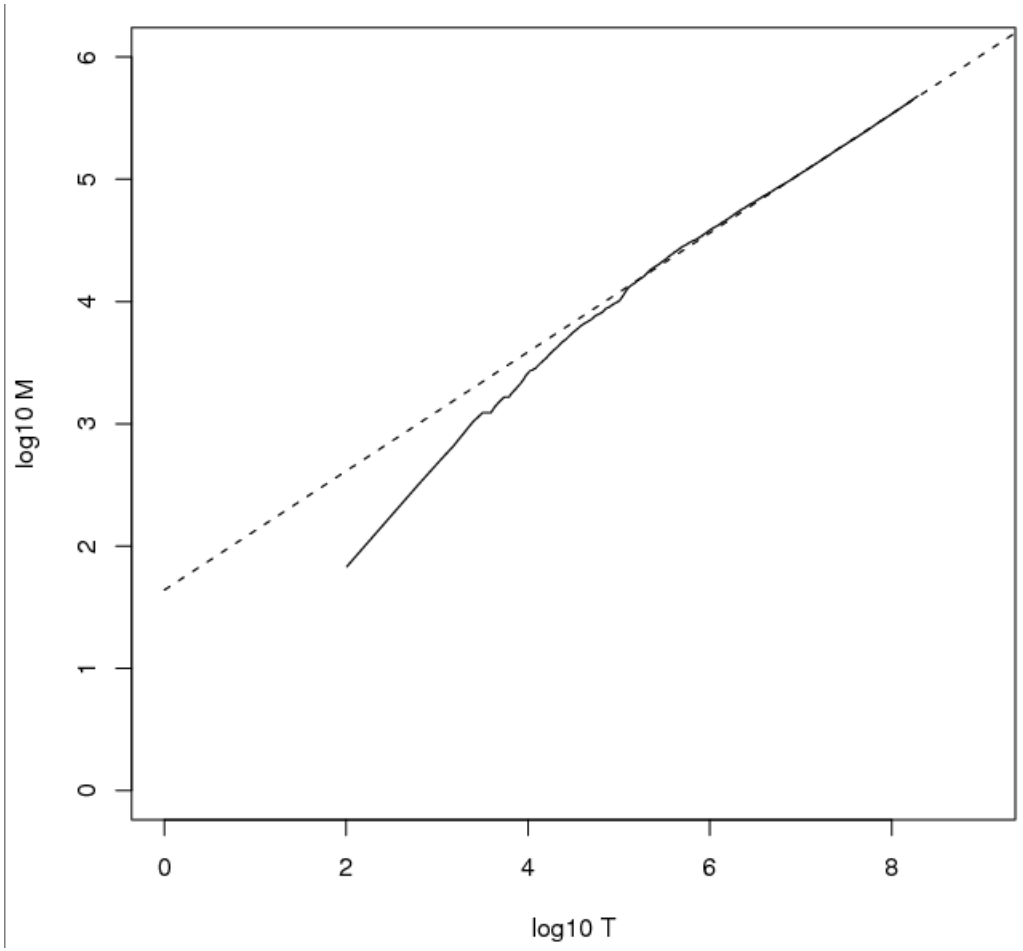
Estimating the number of terms

- **Heaps' law: $M = kT^b$**

- Estimates vocabulary size as a function of the collection size
- M is the size of the vocabulary; T is the number of tokens in the collection,
- Typical values: $30 \leq k \leq 100$ and $b \approx 0.5$
  - It is the simplest possible relationship between the two in log-log space
  - An empirical finding ("empirical law")

Heaps' Law

- For RCV1, the dashed line $\log_{10} M = 0.49 \log_{10} T + 1.64$ is the best least squares fit.
- Thus, $M = 10^{1.64} T^{0.49}$ so $k = 10^{1.64} \approx 44$ and $b = 0.49$.
- Good empirical fit for Reuters RCV1 !
- For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms



Estimating the number of terms

Heaps' law suggests that

(i) the dictionary size continues to increase with more documents in the collection, rather than a maximum vocabulary size being reached

(ii) the size of the dictionary is quite large for large collections.

Zipf's law:Modeling the distribution of terms

- Heaps' law gives the vocabulary size in collections
- We also study the relative frequencies of terms
- In natural language, there are a few very frequent terms and very many very rare terms
- Zipf's law: The ith most frequent term has frequency proportional to $1/i$

$$\mathbf{cf_i \propto 1/i}$$

- cf_i is collection frequency: the number of occurrences of the term t_i in the collection.

Zipf consequences

- **If the most frequent term (*the*) occurs cf_1 times**
  - then the second most frequent term (*of*) occurs $cf_1/2$ times
  - the third most frequent term (*and*) occurs $cf_1/3$ times ...
- Equivalent: $cf_i = K/i$ where K is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$

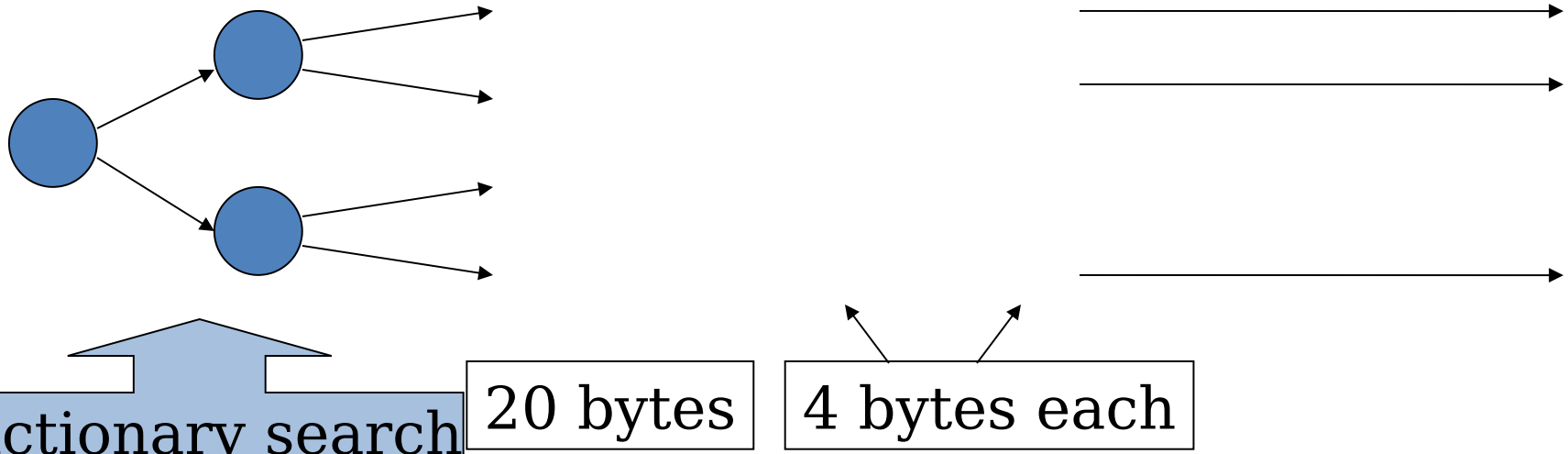
- Another power law relationship

Dictionary Compression

Dictionary storage

Array of fixed-width entries

– ~400,000 terms; 28 bytes/term = 11.2 MB.

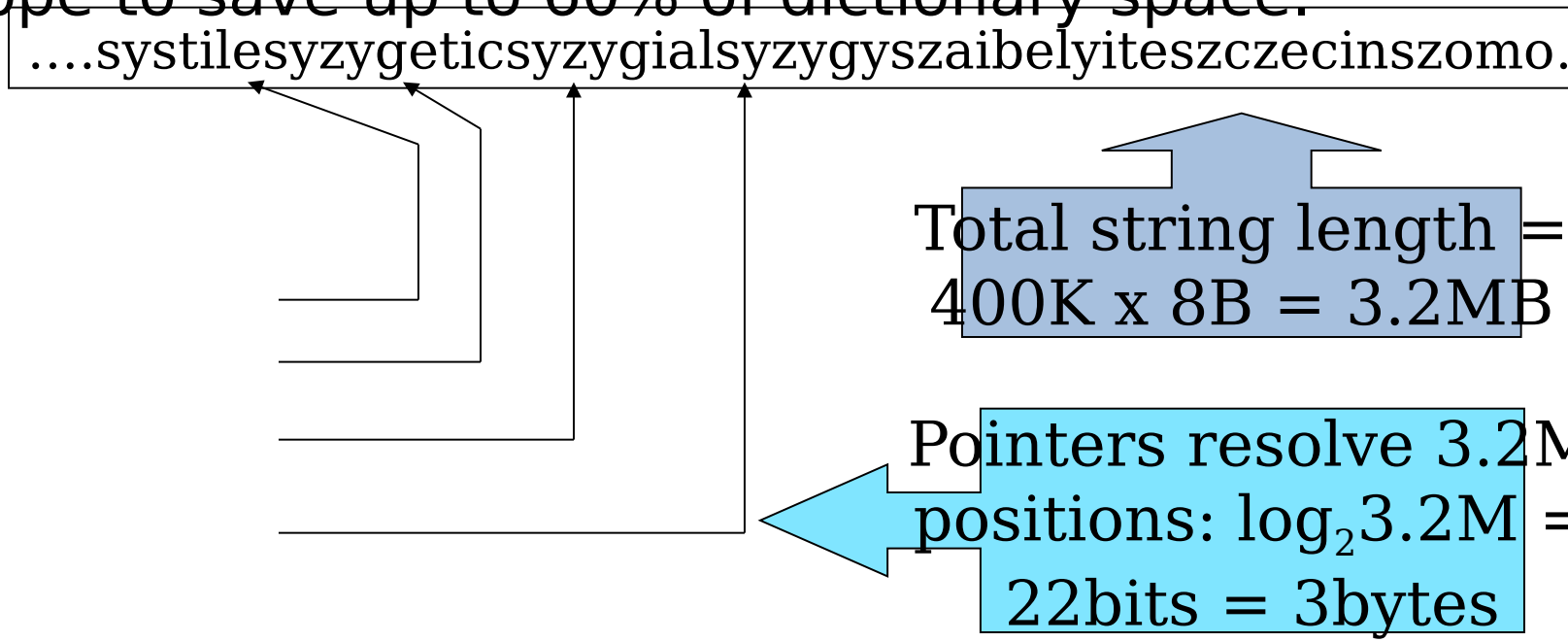


Dictionary storage

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons.*
- Written English averages ~4.5 characters/word.
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

Compressing the term list: Dictionary-as-a-String

- **Store dictionary as a (long) string of characters:**
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space.



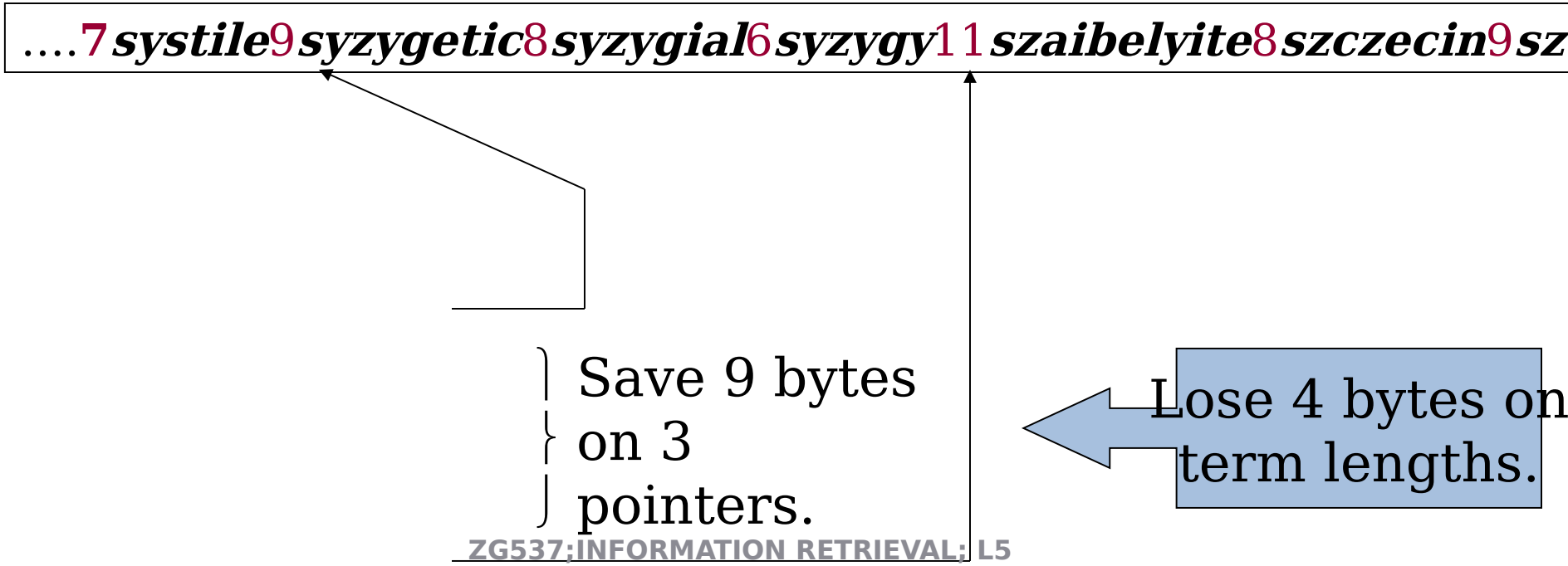
Space for dictionary as a string

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 ⇒ 7.6 MB (against 11.2MB for fixed width)

Now avg. 11 bytes/term, not 20.

Blocking

- Store pointers to every *k*th term string.
  - Example below: *k*=4.
- Need to store term lengths (1 extra byte)


....7systile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9sz
Save 9 bytes on 3 pointers.
Lose 4 bytes on term lengths.

Net

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - 3 x 4 = 12 bytes,
- now we use 3 + 4 = 7 bytes.
- For $k = 4$, we save $(k - 1) \times 3 = 9$ bytes for term pointers, but need an additional $k = 4$ bytes for term lengths.
- The total space requirements for the dictionary of Reuters RCV1 are reduced by 5 bytes per four-term

Saved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
We can save more with larger *k*.

Exercise

Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 8$ and 16.

Exercise

- **Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of *k = 8* and *16.***

Solution:

- K = 8

We will save :(8-1) * 3 = 21 bytes for term pointer

Need additional k =8 for term length so space reduced by 13 bytes per 8 term block

Total space reduced by= 400000 * 13 /8 = 0.65 MB

Total space : 7.6 – 0.65 = 6.95 MB

- K=16

We will save :(16-1) * 3 = 45 bytes for term pointer

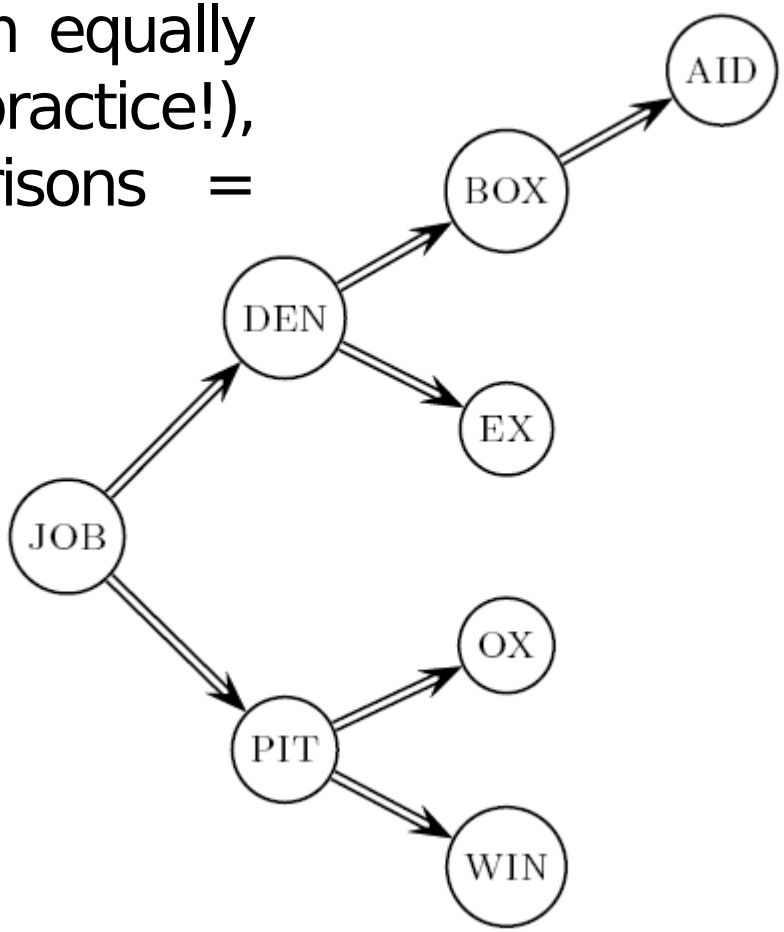
Need additional k=16 for term length so space reduced by 29 bytes per 16 term block

Total space reduced by= 400000 * 29 /16 = 0.725 MB

Total space : 7.6 – 0.725 = 6.875 MB

Dictionary search without blocking

Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = (1+2•2+4•3+4)/8 ~2.6

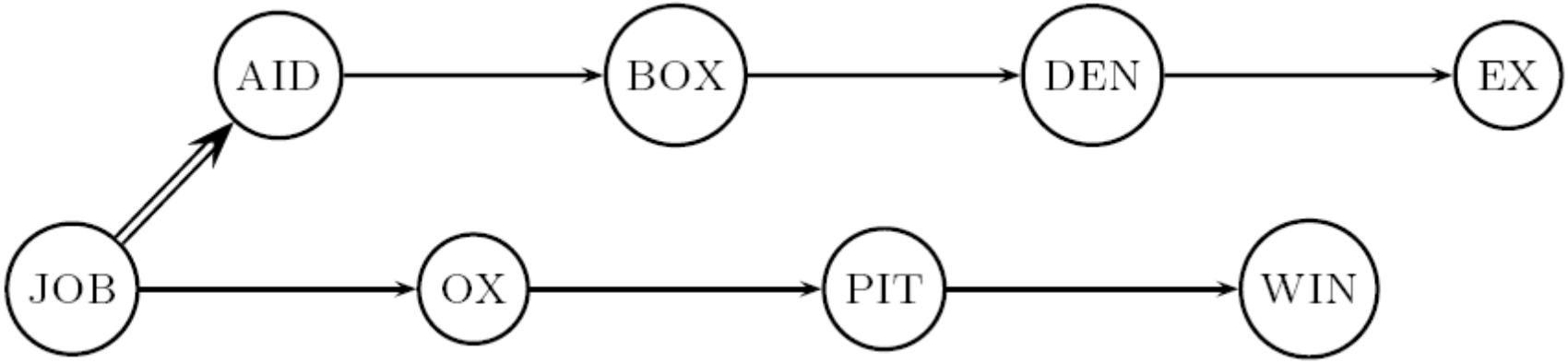


Dictionary search with blocking

Binary search down to 4-term block;

- Then linear search through terms in block.

Blocks of 4 (binary tree), avg. = (1+2·2+2·3+2·4+5)/8 = 3 compares



Front coding

Front-coding:

- Sorted words commonly have long common prefix – store differences only
- (for last *k-1* in a block of *k*)

8***automata***8***automate***9***automatic***10***automation***

→8***automat*******a***1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

Subsequent terms

Begins to resemble general string compression.

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
Also, blocking $k = 4$	7.1
Also, Blocking + front coding	5.9

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10
- Key Point : store each posting compactly
- A posting for our purposes is a docID
- For Reuters (800,000 documents), we would use 32 bits per docID, when using 4-byte integers
- Alternatively, we can use $\log_2$ 800,000 ≈ 20 bits per docID
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2$ 1M ~ 20 bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
  - ***computer***: 33,47,154,159,202 …
- Consequence: it suffices to store *gaps*.
  - 33,14,107,5,43 …
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.

Three postings entries

	encoding	postings list								
THE	docIDs	...		283042		283043		283044		283045 ...
	gaps				1		1		1	...
COMPUTER	docIDs	...		283047		283154		283159		283202 ...
	gaps				107		5		43	...
ARACHNOCENTRIC	docIDs	252000		500100						
	gaps	252000	248100							

Variable length encoding

- Aim:
  - For ***arachnocentric***, we will use ~20 bits/gap entry.
  - For ***the***, we will use ~1 bit/gap entry.
- If the average gap for a term is *G*, we want to use $\sim \log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a *variable length encoding*
- Variable length codes achieve this by using short codes for small numbers

Variable Byte (VB) codes

- For a gap value G, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit c
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode G's lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.

Example

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Exercise

- What is the posting list that can be decoded from the variable byte-code

  10001001 00000011 10000010 11111110 ?

- What is the largest gap that can be encoded in 2 bytes using the variable-byte encoding ?

Exercise

- What is the posting list that can be decoded from the variable byte-code

  10001001 00000011 10000010 11111110 ?

  9 **386** **126**

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, k = 4	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ-encoded	101.0

References

T1 Chapter 5

F. Scholer, H.E. Williams and J. Zobel. 2002. Compression of Inverted Indexes For Fast Query Evaluation. *Proc. ACM-SIGIR 2002*.

- Variable byte codes

V. N. Anh and A. Moffat. 2005. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval* 8: 151–166.

- Word aligned codes
- WDM 56: Various Posting Compression Techniques (first 28 minutes)